Problem Wk.11.1.1: Observation Models

In this problem, we look at defining observation models. The first three problems ask you to define conditional distributions on the color that the robot will observe in a room, given the actual color of that room. Be sure to read the Design Lab 11 handout before working on this problem.

Look at the software documentation for the `dist` module. Throughout this problem, the `dist` module has been imported, so you can get at all the contents by using `dist.x`.

You can assume that the following functions and variables, described in the lab handout, are already defined:

- `makeObservationModel(hallwayColors, obsDist)` -- see lab handout.
- `standardHallway` -- list of the colors of each room in the hallway, going from left to right. In this problem, this is `['white', 'white', 'green', 'white', 'white']`.
- `possibleColors` -- list of all the possible colors that could be observed. In this problem, this is `['black', 'white', 'red', 'green', 'blue']`

In your answers, feel free to define any auxiliary functions that you may need.

Part 1: White == Green

Define an observation noise model (that is, a conditional distribution over the observed color given the actual color of a room) in which white and green are indistinguishable, in the sense that the robot is just as likely to see white as to see green when it is in a white square or a green square, but where all other colors are observed perfectly. This function should return a `dist.DDist` over observed colors, given the actual color passed in as an argument.

```
def whiteEqGreenObsDist (actualColor):
    pass
```

Part 2: White <-> Green

Give an observation noise distribution in which white always looks green, green always looks white, and all other colors are observed perfectly.

```
def whiteVsGreenObsDist (actualColor):
    pass
```

Part 3: Noisy

Define an observation noise distribution in which there is a probability of 0.8 of observing the actual color of a room, and there is a probability of 0.2 of seeing one of the remaining colors (equally likely which other color you see). All possible colors are available in the list called `possibleColors`.

```
def noisyObs(actualColor):
    pass
```

Part 4: Observation Models

The observation noise distributions that you have defined so far encode the error model for observation in a way which is independent of location. But, ultimately, we need to be able to obtain a probability distribution over the possible observations (colors) at a particular location along the hallway. The function `makeObservationModel` allows us to construct such a model, given a list of colors, such as `standardHallway`, and an observation noise distribution, such as you've written above.

Enter the expression for creating the full observation model for the `standardHallway` (assume it's already defined), with the observation noise distribution `noisyObs` (assume it's already defined).

```
noisyObsModel = None
```

Problem Wk.11.1.2: Transition Models

In this problem, we look at defining transition models. It is important to read the Design Lab 11 handout before you start this problem.

Look at the software documentation for the `dist` module. Throughput this problem, the `dist` module has been imported, so you can get at all the contents by using `dist.x`.

In your answers, feel free to define any auxiliary functions that you may need.

The following useful function is already defined:

```
def incrDictEntry(d, k, v):
    if d.has_key(k):
        d[k] += v
    else:
        d[k] = v
```

Part 1: Living on a donut

Write a ``ring'' dynamics model, in which the room 0 is connected to room `hallwayLength-1`.

```
def ringDynamics(loc, act, hallwayLength):
    pass
```

Part 2: Slipping to the left

Write the *left slip* noise model, in which, with probability 0.1, the robot lands one square to the **left** of its nominal location and otherwise lands at its nominal location. It should not be allowed to transition off the end of the world, though: if it is nominally at the leftmost location, it should stay there with probability 1.

```
def leftSlipTrans(nominalLoc, hallwayLength):
    pass
```

Part 3: Noisy Transitions

Write a transition noise model in which the robot lands one square to the left of where it should be with probability 0.1, one square to the right with probability 0.1, and in the nominal square with probability 0.8. It should not be allowed to transition off either end of the world; any probability associated with a square off the end of the hallway should be associated instead with the square at that end of the hallway.

```
def noisyTrans(nominalLoc, hallwayLength):
    pass
```

Part 4: Transition Models

The function `makeTransitionModel`, described in the handout, puts together the dynamics and noise model to construct a full transition model for a hallway of a given length.

Enter the expression for creating the full transition model for the `standardDynamics` and the `noisyTrans` (assume it's already defined) in the `standardHallway` (assume it's already defined) which is 5 squares long.

```
noisyTransModel = None
```

Problem Wk.11.1.4: Simulating Hallways

Below are some hand simulations of state estimation.

Part 1: Perfect sensor, perfect action

Refer to the course notes (Section 7.7). Assume the robot is in a hallway with 3 rooms, with one green room in the middle and a white room on either side. The robot is trying to estimate which room it is in. For this, we will use state estimation with the states being the rooms -- represented as the indices 0 through 2.

We will assume that initially we don't know where the robot is and all states are equally likely.

Let's also assume perfect sensor and motion models, that is, there is no uncertainty in motion or sensing. If the commanded action (one of -1, 0, or 1) would take the robot off the edges of the hallway, then the robot moves just as far as it can and then stops.

Recall that the uppercase S_t, I_t, O_t refer to random variables for the state at time *t*, action at time *t*, and observation at time *t*. Lowercase symbols, *s, i, o* are normal (non-random) variables that denote any value in the domain of states D_s, actions and observations, thus $\sum_{s \in D_s} Pr(S_0 = s) = 1.0$.

We encourage you to do your computations using fractions; you can enter fractions, e.g. 5/27, in the boxes below. If you enter decimals, they need to be accurate to within 0.001.

1. What is the robot's prior belief $B_0(s) = Pr(S_0 = s)$ for each of the states *s*?
   $B_0(s) = Pr(S_0 = s)$

2. First, the robot makes an observation. Let's assume it sees 'white', because it is in a white room and there's no sensor noise. So, $O_0 = white$. We want to know what is the new belief state after this observation?
   - First, figure out $Pr(O_0 = white | S_0 = s)$ for each state *s*
   - Then compute $Pr(O_0 = white | S_0 = s) Pr(S_0 = s)$ for each state *s*. Note that this is the same as $Pr(O_0 = white, S_0 = s)$
   - Now, compute $Pr(O_0 = white)$.
   - Compute the new belief state after the observation; using the definition of conditional probability and the previous two results:
     $B'_0(s) = Pr(S_0 = s | O_0 = white)$.

3. If we told the robot to go right one room with action $I_0 = 1$, what would the belief state be after taking the state transition into account?
   $B_1(s) = Pr(S_1 = s | O_0 = white, I_0 = 1)$

4. Now, assume the robot observes 'green' because it's in a green room and there's no noise, $O_1 = green)$. What will the belief state be after this?
   $B'_1(s) = Pr(S_1 = s | O_0 = white, I_0 = 1, O_1 = green)$

5. If we told the robot to go right with action $I_1 = 1$, what would the belief state be after taking the state transition into account?
   $B_2(s) = Pr(S_2 = s | O_0 = white, I_0 = 1, O_1 = green, I_1 = 1)$

Part 2: Noisy sensor, Perfect action

Refer to the course notes (Section 7.7).

Assume the robot is in a hallway with 3 rooms, with one green room in the middle and a white room on either side. The robot is trying to estimate which room it is in. For this, we will use state estimation with the states being the rooms -- represented as the indices 0 through 2.

We will assume that initially we don't know where the robot is and all states are equally likely.

We'll assume that there are 5 possible colors:

`('black', 'white', 'red', 'green', 'blue')`

We assume a noisy sensor which has a probability of 0.8 of seeing the correct color for the current room and a probability of 0.05 of seeing each of the other posible colors

We assume perfect motion, with actions -1, 0, 1. If the commanded action would take the robot off the edges of the hallway, then the robot moves as far as it can, then stops.

Recall that the uppercase S_t, I_t, O_t refer to random variables for the state at time *t*, action at time *t*, and observation at time *t*. Lowercase symbols, *s, i, o* are normal (non-random) variables that denote any value in the domain of states, actions and observations, thus $\sum_{s \in D_s} Pr(S_0 = s) = 1.0$.

We encourage you to do your computations using fractions; you can enter fractions, e.g. `5/27`, in the boxes below. If you enter decimals, they need to be accurate to within 0.001.

1. What is the robot's prior belief for each of the states *s*?
   $B_0(s) = Pr(S_0 = s)$

2. What is the distribution over what the robot sees? That is, what is $Pr(O_0 = o)$ for all possible colors *o*?
   $P(O_0 = black) =$ $P(O_0 = white) =$ $P(O_0 = red) =$
   $P(O_0 = green) =$ $P(O_0 = blue) =$

3. First, the robot makes an observation. Let's assume it sees 'white'. So, $O_0 = white)$. We want to know the new belief state after the observation. $B'_0(s) = Pr(S_0 = s|O_0 = white)$.

4. If we told the robot to go right $I_0 = 1$, what would the belief state be after taking the state transition into account? Recall that motion is perfect. $B_1(s) = Pr(S_1 = s|O_0 = white, I_0 = 1)$

5. Now, what is the distribution over what the robot sees? That is, what is $Pr(O_1 = o)$ for all possible colors *o*?
$P(O_1 = black) =$ $P(O_1 = white) =$ $P(O_1 = red) =$
$P(O_1 = green) =$ $P(O_1 = blue) =$

6. Now, assume the robot sees 'white' again, $O_1 = white)$. What will the belief state be after this? $B'_1(s) = Pr(S_1 = s|O_0 = white, I_0 = 1, O_1 = white)$

7. If we told the robot to go right $I_1 = 1$, what would the belief state be after taking the state transition into account? $Pr(S_2 = s|O_0 = white, I_0 = 1, O_1 = white, I_1 = 1)$

8. If instead of having seen 'white' and gone right (as above), the robot had seen 'green' and gone right, what would the belief state be? That is, what is $Pr(S_2 = s|O_0 = white, I_0 = 1, O_1 = green, I_1 = 1)$

Problem Wk.11.1.5: Simulating Hallways: The Noisy-Noisy Case

Below are some hand simulations of state estimation.
Don't check every single number you type in! You'll run out of checks and overload the server.

Refer to the course notes (Section 7.7).

Assume the robot is in a one-dimensional grid with 3 squares, with one green square in the middle and white squares on either side. The state indices are 0-based.

We will assume that initially we don't know where the robot is and all states are equally likely.

We'll assume that there are 5 possible colors:

`('black', 'white', 'red', 'green', 'blue')`

We assume a noisy sensor which has a probability of 0.8 of seeing the correct color for the current square and a probability of 0.05 of seeing each of the other posible colors

We assume noisy motion (actions are -1, 0 or +1) as well. The 'nominal' result of taking an action is to move that many spaces in the appropriate direction. If that nominal resulting location is out of bounds, then the nominal location is 'clipped' to be in bounds (that is, it will either be one end or the other of the hallway). With noise, there's a 0.8 probability of moving to the nominal square, and there's a 0.1 probability of being on either side of that square, that is, not moving far enough or going one square too far. If the noisy resulting location is off one end of the hallway, then the the probability associated with that result is assigned to the appropriate hallway end.

Recall that the uppercase S_t, I_t, O_t refer to random variables for the state at time *t*, action at time *t*, and observation at time *t*. Lowercase symbols, *s, i, o* are normal (non-random) variables that denote any value in the domain of states, actions and observations, thus $\sum_{s \in D_s} Pr(S_0 = s) = 1.0$.

We encourage you to do your computations using fractions; you can enter fractions, e.g. `5/27`, in the boxes below. If you enter decimals, they need to be accurate to within 0.001.

1. What is the robot's prior belief $B_0(s) = Pr(S_0 = s)$ for each of the states *s*?
   $B_0(s) = Pr(S_0 = s)$

2. First, the robot makes an observation. Let's assume it sees 'white'. So, $O_0 = white$. We want to know the new belief state after the observation $B_0'(s) = Pr(S_0 = s | O_0 = white)$.

3. If we told the robot to go right $I_0 = 1$, what would the belief state be after taking the state transition into account?
   $B_1(s) = Pr(S_1 = s | O_0 = white, I_0 = 1)$

4. Now, assume the robot sees 'white' again, $O_1 = white$. What will the belief state be after this?
$B'_1(s) = Pr(S_1 = s | O_0 = white, I_0 = 1, O_1 = white)$

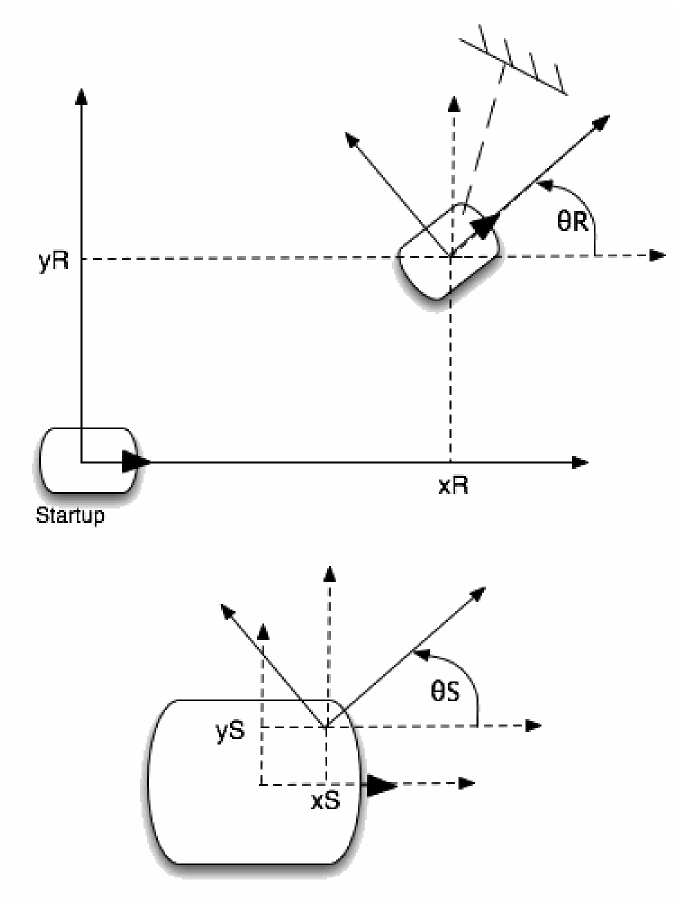
5. If we told the robot to go right $I_1 = 1$, what would the belief state be after taking the state transition into account?
$B_2(s) = Pr(S_2 = s | O_0 = white, I_0 = 1, O_1 = white, I_1 = 1)$

Problem Wk.11.1.6: Sonar hit

One computation that comes up when interpreting the sonar sensors on the robots is the following: Given a distance measurement from one of the sonars, return an instance of `Point` (look at the documentation of the `util` module), in the global odometry frame, the same coordinate frame that the robot's pose is measured, representing where the sonar beam bounced off an object. To do this, we make the assumption that the sonar beam is a line segment emanating from the sonar sensor.

To compute this, we need to know the location and orientation (an instance of `Pose`) of the sonar on the robot and we need to know the Pose of the robot.

- The sonar location on the robot is given by `Pose(xS, yS, thetaS)` where, `xS` and `yS` are the center of the sensor **(relative to the center of the robot)** and `thetaS` is the angle that the beam makes to the robot's heading (the direction the robot's nose points to).
- The robot's pose is `Pose(xR, yR, thetaR)`, as described in the Lab Infrastructure Guide.



It is useful when computing this to think about first finding the location of the hit point relative to the robot and then computing the position of that point relative to the global odometry frame.

Here's a useful bit of math. Imagine you have two coordinate frames, call them A and B. The origin of B is at location (xB, yB), relative to A, and the x-axis of B is rotated by thetaB relative to the x-axis of A. Then, if we we have a point with coordinates bx and by relative to coordinate frame B, we can find the coordinates ax and ay of that point relative to A as follows:

ax = xB + cos(thetaB)*bx - sin(thetaB)*by
ay = yB + sin(thetaB)*bx + cos(thetaB)*by

Note that when thetaB is zero, this says that ax = xB + bx and ay = yB + by, which is what we would expect. Look at the documentation for `Pose.transformPoint` in module `util`; it transforms the Point by displacing it by pose.x and pose.y and rotating it by pose.theta.

For debugging, you might find it useful to draw a picture of the test cases so as to understand what the answer is supposed to be.

Write the function `sonarHit` that is given a distance measurement from one of the sonars, the sonar's pose on the robot and the robot's pose. It should return an instance of `Point` (in the global odometry frame, the same coordinate frame that the robot's pose is measured) representing where the sonar beam bounced off an object.

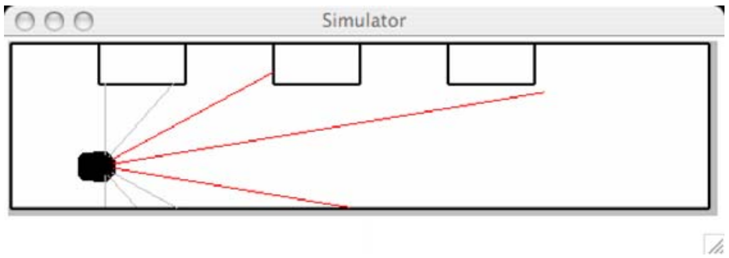
You need to specify `util.` to get functions and classes from `util`.

```
def sonarHit(distance, sonarPose, robotPose):
    pass
```

Problem Wk.11.1.7: Ideal Sonar Readings

We call the problem of figuring out where a robot is in the world, given a known map, but unknown starting location, the problem of **localization**. We can frame the problem as one of doing state estimation. If we specify a model of how our robot interacts with the world, in the form of the definition of a *ssm.StochasticSM*, and feed that into the state estimator, then the state estimator will continually update a probability distribution representing its beliefs about the location of the robot given the actions and observations taken so far.

In Design Lab 13 we are going to build a simple system in which the robot starts out facing to the right, and moves along a straight line to the right, using just the values of its left (number 0) sonar sensor to get information.



Part of the robot model is an error model for the sonar readings. Our error model for sonar readings will have a similar structure to the error model for observing colors in the hallway: we determine the sonar observation we would expect to see in each state, if our observations were perfect, and then use that as the basis for constructing a distribution over possible observations.

In this problem, we consider how to compute the ideal sonar readings. We are given:

- `wallSegs`: A list of line segments representing the walls in the simulated world. Each line segment is an instance of the `util.LineSeg` class (look at the software documentation for this class).
- `robotPoses`: A list of poses (instances of `util.Pose` class) of the robot, corresponding to the center of each of the discretized robot states.
- `sonarHit(distance, sonarPose, robotPose)`: a procedure that takes the length of a line segment emanating from a sonar sensor (`dist`), the pose of the sensor in the robot's frame (represented as a `util.Pose`), and the pose of the robot in the global frame (represented as a `util.Pose`), and returns a `util.Point` representing the position of the end of the sonar segment in the global frame. This function is already defined.
- `sonarPose0`: the pose of sonar sensor 0 with respect to the robot is given as an instance of `util.Pose`.
- `sonarMax`: the maximum meaningful value of a sonar reading; any distance longer than this should be represented as `sonarMax`.
- `numObservations`: the sonar readings (from 0 to `sonarMax`) will be discretized into integers between 0 and `numObservations-1`.

Implement two procedures:

1. `discreteSonar` discretizes sonar readings: it takes a sonar reading as input and generates an integer bin index as ouput. Sonar readings will range between 0 and `sonarMax`. This range should be discretized into a fixed number, `numObservations`, of bins. Note that the bin indices are from 0 to `numObservations-1`. Any sonar reading

greater than `sonarMax` should be placed into the last bin.

We strongly suggest that you draw a picture, with the interval from 0 to 1 divided into three bins. Consider what answers you would want to return for inputs of 0.32 and 0.34. Here are some useful things to know about Python:

- The `round` procedure takes a floating point number and returns the nearest whole number, as a float. So, `round(2.8)` returns `3.0` and `round(2.1)` returns `2.0`.
- The `int` procedure takes a floating point number and returns just its integer part, as an integer. So, `int(2.1)` returns `2`.

2. `idealReadings` takes a list of wall segments describing the world and a list of robot poses, corresponding to each of the states of the system, and returns **a list of discretized ideal sonar readings for sonar 0, one for each state**. For each state:
   - Determine the line segment starting at the point where sonar sensor 0 is currently located in the world, and going out in the direction of the sensor for length `sonarMax`.
   - Find the point of intersection between that line segment and any wall segment in the world that is closest to sonar sensor 0.
   - Compute the ideal sonar reading, which is the distance from the sensor to that closest point of intersection. You can use the `intersection` method of `util.LineSeg`, which returns the `util.Point` of intersection between two line segments, if there is one; it returns `False` if there isn't one. If there is no intersection, then the ideal sonar reading is `sonarMax`.
   - Compute a discretized value for the ideal sonar reading.

   Return the list of discretized ideal readings, one for each state.

Debug this in Idle and paste your answer here.

One test case is:

```
def wall((x1, y1), (x2, y2)):
   return util.LineSeg(util.Point(x1,y1), util.Point(x2,y2))
wallSegs = [wall((0, 2), (8, 2)),
            wall((1, 1.25),(1.5, 1.25)),
            wall((2, 1.75),(2.8, 1.75))]
robotPoses = [util.Pose(0.5, 0.5, 0), util.Pose(1.25, 0.5,0),
              util.Pose(1.75, 1.0, 0), util.Pose(2.5, 1.0, 0)]
```

```
sonarMax = 1.5
numObservations = 10
sonarPose0 = util.Pose(0.08, 0.134, 1.570796)

def discreteSonar(sonarReading):
    pass

def idealReadings(wallSegs, robotPoses):
    pass
```